Version: 1.3
Date: January 07
Authors: Olivier Colinet, Paul Thompson,
César Lázaro (v1.3 update)

# GSA HTTP Sniffer utility

**Change History**

Version 1.0

- Initial release

Version 1.1

- Support MULTIPART method to upload files or parameters. Can be used with the Google Search Appliance to script the import of configuration or to create collections, which use multipart methods. Sample Sniffer configuration file included - gsaImportConfig.xml and AddCollection.xml
- Support for SAVEAS to save the output of a specific request to a file. Sample Sniffer configuration file included gsaExportConfig.xml
- Reading variable input from a file, ability to define a variable's values(s) to be read from a file. Can be used to added a list of start URL's from a file and add them to the GSA Admin Start
- URL. Sample Sniffer configuration file included - AddCrawlURL.xml
- bug fixes

  Fixed problem with spaces in path in sniffer.xsd file

Version 1.2

- Added support for HTTP sites that do not have a valid certificate. Use -Dssl.cert=any to disable any certificate validation. Useful for sites that user self certification; option should be used with caution.
- Removed the need to specify the location of sniffer.xsd, it is assumed to be in `conf` directory

Version 1.3

- SPNEGO (Kerberos) authentication mechanism has been added to the other ones already supported in previous versions. New Kerberos credentials classes have been included.
- bug fixes

  Fixed issue when sending cookies over to sendRequest method
  Lack of scalability solved using configurable connection pooling parameters
  Fixed problem with query strings

## 1 Introduction

In today's environment, the Enterprise system Web front-ends often make use of JavaScript event handlers and browser-specific customizations. Their integration with commercial and proprietary Single Sign On systems may bring its own additional set of Enterprise-specific customizations that are sometimes very difficult to identify for a non security expert.

The HTTP Sniffer was designed to bridge the gap and diagnose any HTTP communication "from the inside". The HTTP Sniffer is first and foremost a configurable HTTP client that can automate the execution of a Web process, i.e. a set of HTTP or HTTPs requests. It provides tracing capabilities for any HTTP headers, parameters and cookies and handles in standard most redirects, certificates and authentication schemes.

More importantly any HTML content accessed successfully by the HTTP Sniffer is cached locally so that the end-user can isolate easily any interfering JavaScript snippet and reproduce accordingly its actions. By incrementally configuring a set of HTTP requests, the end-user will achieve successfully the automation of a secure or public Web process.

The HTTP Sniffer utility will help system administrators, security experts, Webmasters, developers to diagnose and understand a specific set of HTTP or HTTPs requests. It will assist in identifying any *User-Agent* filtering or any interfering JavaScript snippet that are usually almost impossible to isolate with standard utilities (Firefox liveHttpHeaders and the like). The HTTP Sniffer will not reduce the complexity of the on-going HTTP communication (i.e., programmatic and HTTP redirects, etc.) between several Web-enabled systems but will certainly contribute in providing more visibility on the underlying infrastructure.

## 2 A configurable HTTP client

A Web process is described in a well-formed and valid XML file (for example, *web_process.xml*) and executed by the following command

```
sniffer.bat|sh <path to web process xml>.
```

The `sniffer.xsd` file that comes with the utility controls the grammar of the Web process descriptor.

Each HTTP request targeting a certain URL may be of three kinds (*GET*, *POST* and *HEAD*) and may include a set of HTTP headers and/or parameters. As a convenience, the HTTP Sniffer grammar provides the ability to set global headers shared by all requests of a Web process. This may particularly be useful to set a specific *User-Agent*, etc.

The example below illustrates a Web process called google that accesses the Google homepage and sends a simple query by entering the '`google search appliance`' keywords. It makes use of a global header that sets the `User-Agent` value to be the one of the Firefox browser; and a few HTTP parameters that define the Google query context.

```
<?xml version="1.0"?>

<webProcess name="google" xmlns="http://www.google.com/enterprise/gsa">
      <header name="User-Agent" value="Mozilla/5.0 (Windows; U; Windows NT
            5.1; en-US; rv:1.8) Gecko/20051111 Firefox/1.5"/>
      <request>
            <type>GET</type>
           <URL>http://www.google.com/</URL>
      </request>
      <request>
            <type>GET</type>
           <URL>http://www.google.com/search</URL>
            <parameter name="hl" value="en"/>
           <parameter name="q" value="google search appliance"/>
           <parameter name="btnG" value="Google Search"/>
      </request>
</webProcess>
```

The `logs/sniffer.log` file contains all information about the HTTP request headers passed by the end-user, the HTTP response headers returned by the Web server, the HTTP cookies set by the Web server and the corresponding HTTP request status (message and code).

Upon completion of a Web process, a zipped file is created in the directory `tmp/`. Its name is of the following pattern: `process_name-date.zip` where `process_name` is the name set in the XML descriptor and `date` is of type `HHmmss`. This zipped file contains a copy of the `sniffer.log` file and of all HTML pages accessed by the HTTP Sniffer (e.g. `requestSnifferX.html` where Xis the sequence number of the HTTP request in the process description) along the execution of the Web process.

In the case where a Web process would contain several HTTP calls and an HTTP parameter value would be set (by some JavaScript code) in an HTML page and required in the subsequent HTTP request(s), a copy of the latest HTML page accessed by the HTTP Sniffer (e.g. `requestSnifferX.html` where Xis the sequence number of the request in the process description) is also saved in the `Sniffer_root_dir\tmp\` directory.

## 3 An interactive HTTP client

In order to provide user interactivity along with the execution of a Web process, the concept of variables is available.
Variables are simple placeholders.
Two types of variables are currently support, user input and file

**User Input**

They are set dynamically by the end-user from a prompt shell. Their format can be controlled by a regular expression and a default value may be assigned. Any header, cookie and parameter name and/or value can be assigned dynamically via a variable. The URL of an HTTP request can also be set dynamically.
As illustrated below, a *keywords* variable (with no default value) has been defined and allows the end-user to enter from his console a list of keywords. The regular expression assigned to the variable controls the user input format.

```
<variable name="keywords" default="" type="([a-z])*"/>

<request>
      <type>GET</type>
      <URL>http://www.google.com/search</URL>
      <parameter name="hl" value="en"/>
      <parameter name="q" value="{$keywords}"/>
      <parameter name="btnG" value="Google Search"/>
</request>
```

As a direct consequence, variables may be set within a Web process descriptor to intercept a value of a request parameter (assigned dynamically by some JavaScript code on page A and required for subsequent request B), to assign an *Authorization* header returned by a successful authentication request, and/or to avoid sensitive information (password, login) to be entered in clear.

**File Input**

Used to point to a file contain the contents to be used for the variable. See the XML configuration AddCrawlUrl's for an example usage of the file input, where a list of Start Urls is read from a file.
The sample below makes use of a file to source the contents of a variable, in this example the list of start URL's are listed in a file:

```
start.txt
http://www.google.com/enterprise
http://support.google.com/
```

Sample XML snippet using file variable type

```
<request>
  <type>GET</type>
  <URL>http://GSA_HOSTNAME:8000/</URL>
</request>

<file name="startUrlsFile" src="start.txt"/>
<file name="goodUrlsFile" src="include.txt"/>
<file name="badUrlsFile" src="exclude.txt"/>
```

**Downloading files**

Even though all communication between the Sniffer and the web server is logged, there is sometimes a need to save the results of a HTTP request into a separate file. This is done by using the *saveAs* parameter and providing a valid file name. Here is an example where we have added this option to the above web process:

```
<!-- export -->
```

```
<request>
      <type>GET</type>
      <saveAs>gsaExportConfig.xml</saveAs>
      <URL>http://GSA_HOSTNAME:8000/EnterpriseController</URL>
      <parameter name="actionType" value="importExport"/>
      <!--  export paraphrase -->
      <parameter name="password1" value="exportpassword"/>
      <parameter name="password2" value="exportpassword"/>
      <parameter name="export" value=" Export Configuration "/>
</request>
```

This feature is particularly useful when using the sniffer for automating administrative tasks, e.g. it can be used to download the configuration file from the Google Search Appliance and thus creating a configuration backup routine.

**Uploading files**

Uploading files is done with by using the *MULTIPART* request type in conjunction with the *file* parameter. The *MULTIPART* request type work as a normal *POST* but will add an extra part to the HTTP post by reading, embedding and sending the file specified by the *file* parameter. It could look like this:

```
<!-- import -->
  <request>
    <type>MULTIPART</type>
    <URL>http://GSA_HOSTNAME:8000/EnterpriseController</URL>
    <parameter name="actionType" value="importExport"/>
    <parameter name="passwordIn" value="{$passwordIn}"/>
    <parameter name="import" value="Import Configuration"/>
      <parameter name="password1" value=""/>
      <parameter name="password2" value=""/>
    <file name="importFileName" value="gsaConfig.xml"/>
  </request>
```

This configuration would treat `passwordIn` as a variable and perform a file upload from the file at `importFileName`.

## 4 HTTP authentication schemes

The HTTP Sniffer supports all standard authentication schemes (HTTP Basic, HTTP Digest, SPNEGO/Kerberos and NTLM).
Upon successful HTTP authentication (i.e. HTTP challenge), an `Authorization` request header is set and can be reused by all subsequent requests.

The value of the `Authorization` header can be read in the `sniffer.log` file passed consequently as a variable to all following requests.

As an example, the following Web process automates the secure search on a Google Search Appliance where content is protected with HTTP BASIC authentication. The first request attempts to perform the search and passes the users credentials via variables. Provided that the end-user is authorized to search an `Authorization` header is set upon successful authentication challenge recorded in the log file.

```
2006-02-07 10:58:18,942 [Thread-0] INFO - HTTP response header: WWW-Authenticate : Basic
realm="Google Authentication"
```

When prompted for the `authorization` variable, the value of the header from the first request can copy the `Authorization` header value and enter the keywords for the second query.

```
<webProcess name="moma" xmlns="http://www.google.com/enterprise/gsa">
      <header name="User-Agent" value="Mozilla/5.0 (Windows; U; Windows NT
            5.1; en-US; rv:1.8) Gecko/20051111 Firefox/1.5"/>
      <variable name="keywords" default="" type="([a-z])*"/>
      <variable name="username" default="userID" type=""/>
      <variable name="password" default="" type=""/>

      <request>
            <authentication>
                  <Basic>
                        <username>{$username}</username>
                        <password>{$password}</password>
                  </Basic>
            </authentication>
            <type>GET</type>
      <URL>http://gsa.domain/search</URL>
            <parameter name="q" value="{$keywords}"/>
            <parameter name="site" value="internal"/>
            <parameter name="client" value="internal"/>
            <parameter name="output" value="xml_no_dtd"/>
            <parameter name="ie" value="UTF-8"/>
            <parameter name="oe" value="UTF-8"/>
            <parameter name="proxystylesheet" value="internal"/>
            <parameter name="access" value="a"/>
      </request>

      <variable name="authorization" default="" type=""/>
      <variable name="keywords" default="" type="([a-z])*"/>

      <request>
      <type>GET</type>
      <URL>http://gsa.domain/search</URL>
      <header name="Authorization" value="{$authorization}"/>
      <parameter name="q" value="{$keywords}"/>
            <parameter name="site" value="internal"/>
            <parameter name="client" value="internal"/>
            <parameter name="output" value="xml_no_dtd"/>
            <parameter name="ie" value="UTF-8"/>
            <parameter name="oe" value="UTF-8"/>
      <parameter name="proxystylesheet" value="internal"/>
      <parameter name="access" value="a"/>
       </request>
</webProcess>
```

When using SPNEGO authentication mechanism, it requires some special variables that configure the Kerberos environment so that the configuration file looks like different. The sniffer manages Kerberos tickets in a couple of ways:

- They are created on the fly using username and password credentials. This configuration is pretty similar to the `authentication` tag shown in the example above, changing `Basic` to `Kerberos`.
- They are obtained from a special Kerberos file, keytab that contains the ticket. The configuration information is stored in a special Java configuration file. An example (`spnego.conf`) of this file looks like:

```
com.sun.security.jgss.initiate {
    com.sun.security.auth.module.Krb5LoginModule required
    principal=HTTP/server.google.com@ENTERPRISE.GOOGLE.COM
    keytab="C:/keytabFileLocation/krb5.keytab"
    useKeyTab=true
    storeKey=true
    doNotPrompt=true
    debug=true;
};
```

This file points to the already created keytab file and also specifies its principal service name that in this case is *HTTP/server.google.com* in the *ENTERPRISE.GOOGLE.COM* Kerberos domain. The Kerberos service name ticket has to be registered in the KDC (Key Distribution Center) that in Windows environments coincide with the Domain Controller. The keytab file is created by the Kerberos' `ktpass` utility. Check how to create them as it really depends on your platform.

The Kerberos v5 configuration file also has to be present. This file is usually name as `krb5.ini` in Windows whereas in Linux this file is usually located at `/etc/krb5.conf`.

You can find some configuration examples at the Sniffer `conf` directory.

## 5 HTTP cookies

The values of the HTTP cookies set by the Web servers are recorded in the `sniffer.log` file (see the *Set-Cookie* response headers). Since some authentication cookies are set by all Single Sign On systems, it may be important in certain cases to test the authentication process and the authorization process separately. Different HTTP connections may handle the authentication process and the authorization process, which in our case would consist in writing two different XML process descriptors (the cookies are forwarded from one HTTP connection to another as long as they are initiated in the same domain).

In order to do so, the `sniffer.xsd` grammar provides the ability to set the cookies manually. By executing the Web process described in the section above, HTTP Authentication Schemes, two cookies were set by the Web server.

```
2006-02-07 10:17:16,535 [Thread-0] INFO - HTTP cookie : COOKIETEST=1; path=/; secure
2006-02-07 10:17:16,535 [Thread-0] INFO - HTTP cookie : AUTH_26335425=3hahkf9sho62l6zu;
path=/search; domain=gsa.domain; secure
```

These cookies will be valid for a certain period of time (until they expire).

The following example illustrates how Search authentication cookies may be set manually from the Web process descriptor and how the authorization process may be tested separately. Since there is no need for authenticating again, the HTTP request is declared as public. Until expiration of the cookies, some search results will be returned successfully.

```
<?xml version="1.0"?>

<webProcess name="moma" xmlns="http://www.google.com/enterprise/gsa">
      <cookie domain="" name="COOKIETEST" value="1" secure="true"/>
      <cookie domain="gsa.domain" name="AUTH_26335425"
            value="3hahkf9sho6216zu" path="/search" secure="true"/>

      <header name="User-Agent" value="Mozilla/5.0 (Windows; U; Windows NT
            5.1; en-US; rv:1.8) Gecko/20051111 Firefox/1.5"/>

      <variable name="keywords" default="" type="([a-z])*"/>

      <request>
            <type>GET</type>
            <URL>http://gsa.domain/search</URL>
            <parameter name="q" value="{$keywords}"/>
            <parameter name="site" value="internal"/>
            <parameter name="client" value="internal"/>
            <parameter name="output" value="xml_no_dtd"/>
            <parameter name="ie" value="UTF-8"/>
            <parameter name="oe" value="UTF-8"/>
            <parameter name="proxystylesheet" value="internal"/>
            <parameter name="access" value="a"/>
      </request>
</webProcess>
```

## 6 System requirements

The HTTP Sniffer utility is a Java program and as such requires a local JRE (1.5.0 and upwards). It depends upon the following libraries:

commons-collections-3.2.jar
- http://jakarta.apache.org/commons/collections/

commons-httpclient-3.0.1.jar
- http://jakarta.apache.org/commons/httpclient/

commons-logging-1.1.jar
- http://jakarta.apache.org/commons/logging/

commons-codec-1.3.jar
- http://jakarta.apache.org/commons/codec/

log4j-1.2.14.jar
- http://logging.apache.org/log4j/docs/index.html

jdom.jar
- http://www.jdom.org/

xercesImpl.jar
- http://xerces.apache.org/xerces-j/